Rutgers University
School of Engineering

Fall 2011

14:440:127 - Introduction to Computers for Engineers

Sophocles J. Orfanidis
ECE Department
orfanidi@ece.rutgers.edu

week 9

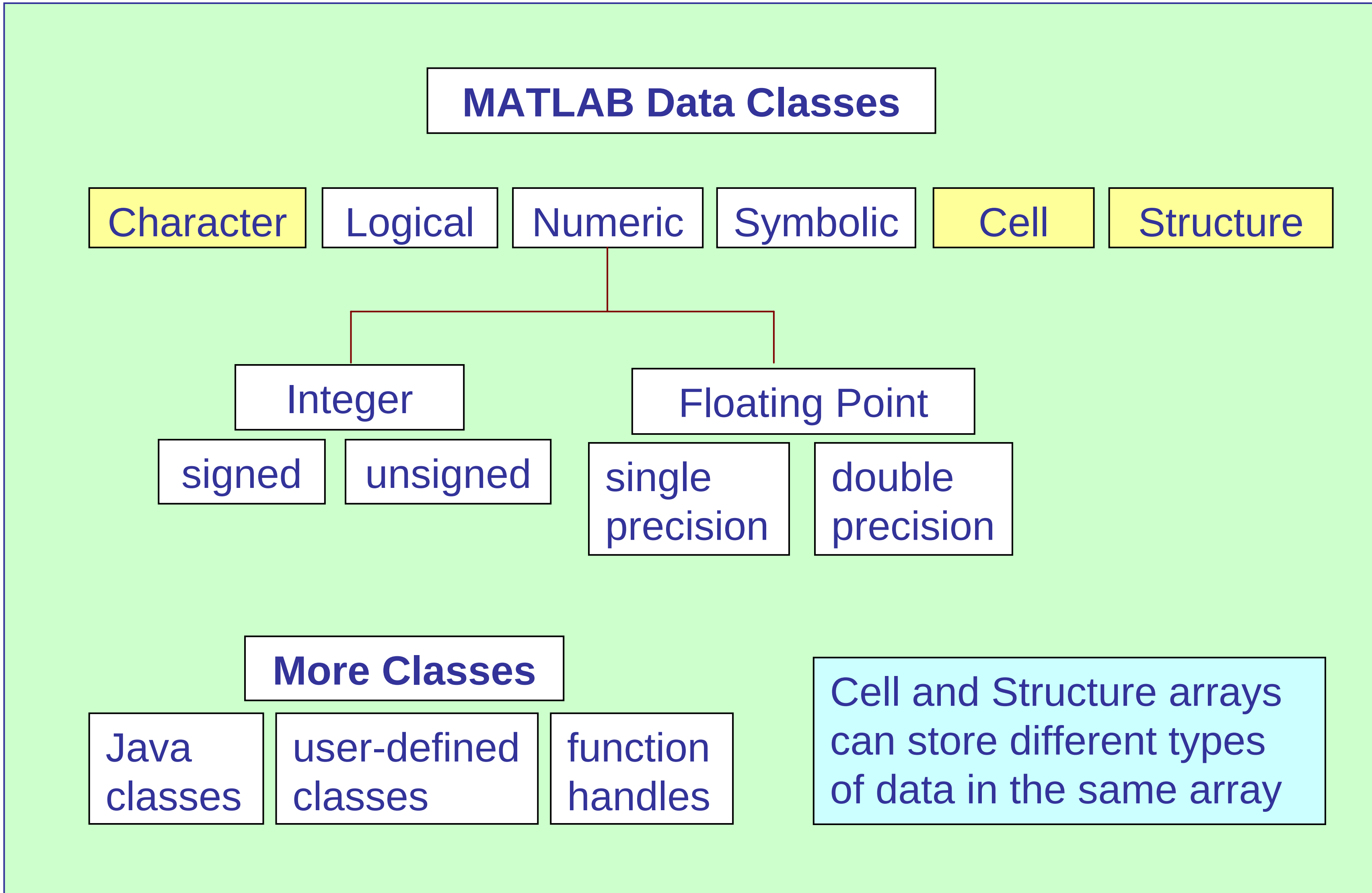
Weekly Topics

Week 1 - Basics – variables, arrays, matrices, plotting (ch. 2 & 3)
Week 2 - Basics – operators, functions, program flow (ch. 2 & 3)
Week 3 - Matrices (ch. 4)
Week 4 - Plotting – 2D and 3D plots (ch. 5)
Week 5 - User-defined functions (ch. 6)
Week 6 - Input-output processing (ch. 7)
Week 7 - Program flow control & relational operators (ch. 8)
Week 8 - Matrix algebra – solving linear equations (ch. 9)
Week 9 - Strings, structures, cell arrays (ch. 10)
Week 10 - Symbolic math (ch. 11)
Week 11 - Numerical methods – data fitting (ch. 12)
Week 12 – Selected topics

Textbook: H. Moore, *MATLAB for Engineers*, 2nd ed., Prentice Hall, 2009

Strings, Structures, Cells

- characters and strings
- concatenating strings
- using **`num2str`**
- comparing strings with **`strcmp`**
- structure arrays
- converting structures to cells
- cell arrays
- cell vs. content indexing
- **`varargin, varargout`**
- multi-dimensional arrays


MATLAB Data Classes
Character
Logical
Numeric
Symbolic
Cell
Structure
Integer
signed
unsigned
Floating Point
single precision
double precision
More Classes
Java classes
user-defined classes
function handles
Cell and Structure arrays can store different types of data in the same array

Characters and Strings

Strings are arrays of characters.

Characters are represented internally by standardized numbers, referred to as ASCII (American Standard Code for Information Interchange) codes.
see Wikipedia link: ASCII table

```
>> c = 'A'
c =
A

>> x = double(c)
x =
    65   % ASCII code for 'A'

>> char(x)
ans =
A

>> class(c)
ans =
char
```

`char()` creates a character string

```
>> doc char
>> doc class
```

```
>> s = 'ABC DEFG'
s =
ABC DEFG

>> x = double(s)
x =
    65 66 67 32 68 69 70 71

>> char(x)
ans =
ABC DEFG

>> size(s)
ans =
     1     8
>> class(s)
ans =
char
```

ASCII codes

convert ASCII codes to characters

s is a row vector of 8 characters

```
>> s(2), s(3:5)
ans =
B
ans =
C D
```

Concatenating Strings

```
s = ['Albert', 'Einstein']
s =
AlbertEinstein

>> s = ['Albert', ' Einstein']
s =
Albert Einstein

>> s = ['Albert ', 'Einstein']
s =
Albert Einstein

>> size(s)
ans =
     1    15
```

preserve leading and trailing spaces

```
>> doc strcat
>> doc strvcat
>> doc num2str
>> doc strcmp
>> doc findstr
```

Concatenating Strings

```
s = strcat('Albert ', 'Einstein')
s =
AlbertEinstein
```

strcat strips trailing spaces but not leading spaces

```
>> s = strcat('Albert', ' Einstein')
s =
Albert Einstein
```

use **repmat** to make up long format strings for use with **fprintf**

```
>> fmt = strcat(repmat('%8.3f  ',1,6),'\n')
fmt =
%8.3f  %8.3f  %8.3f  %8.3f  %8.3f  %8.3f\n
```

Concatenating Vertically

```
s = ['Apple'; 'IBM'; 'Microsoft'];

??? Error using ==> vertcat
CAT arguments dimensions are not consistent.

s = ['Apple    '; 'IBM      '; 'Microsoft']

s =
Apple
IBM
Microsoft

>> size(s)
ans =
     3     9
```

padded with spaces to the length of the longest string

Concatenating Vertically

```
s = strvcat('Apple', 'IBM', 'Microsoft');

s = char('Apple', 'IBM', 'Microsoft');

s =
Apple
IBM
Microsoft

>> size(s)
ans =
     3     9
```

`strvcat,char`
both pad spaces as necessary

Recommendation: use **`char`** to concatenate vertically, and **`[]`** to concatenate horizontally

```
a = [143.87, -0.0000325, -7545]';
```

num2str

```
>> s = num2str(a)
s =
    143.87
-3.25e-005
     -7545
```

```
s = num2str(A)
s = num2str(A, precision)
s = num2str(A, format)
```

```
>> s = num2str(a,4)
s =
     143.9
-3.25e-005
     -7545
```

max number of digits

```
>> s = num2str(a, '%12.6f')
s =
  143.870000
   -0.000032
-7545.000000
```

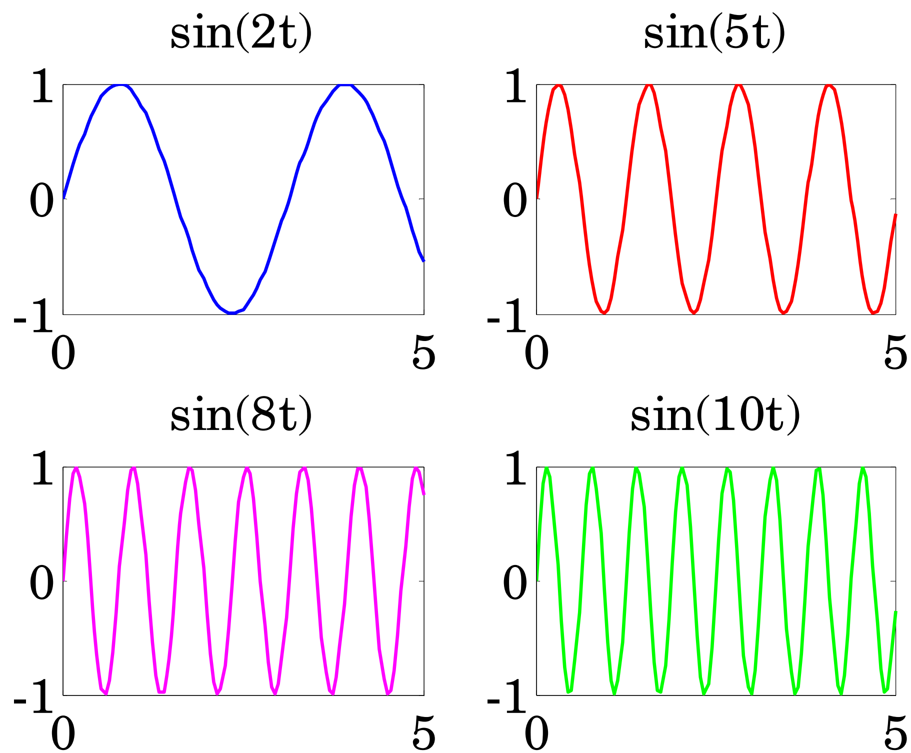
num2str

```
a = [143.87, -0.0000325, -7545]';

>> s = num2str(a, '%10.5E')
s =
 1.43870E+002
-3.25000E-005
-7.54500E+003

b = char('A', 'BB', 'CCC');

>> disp([b, repmat('  ',3,1), s])
A      1.43870E+002
BB    -3.25000E-005
CCC   -7.54500E+003
```

```
t = linspace(0,5,101); w = [2,5,8,10];
Y = sin(w'*t);     % 4x101 matrix
s = char('b-', 'r-', 'm-', 'g-');
for i=1:4,
  figure; plot(t,Y(i,:), s(i,:));
  title(['sin(', num2str(w(i)), 't)']);
  print('-depsc', ['file',num2str(i),'.eps']);
end
```

Labeling and saving multiple plots with **`num2str`**

saved as **`file1.eps, file2.eps file3.eps, file4.eps`**



Comparing Strings

Strings are arrays of characters, so the condition **s1==s2** requires both **s1** and **s2** to have the same length

```
>> s1 = 'short'; s2 = 'shore';
>> s1==s1
ans =
     1     1     1     1     1

>> s1==s2
ans =
     1     1     1     1     0

>> s1 = 'short'; s2 = 'long';
>> s1==s2
??? Error using ==> eq
Matrix dimensions must agree.
```

Comparing Strings

Use **`strcmp`** to compare strings of unequal length, and get a binary decision

```
>> s1 = 'short'; s2 = 'shore';
>> strcmp(s1,s1)
ans =
     1
>> strcmp(s1,s2)
ans =
     0
>> s1 = 'short'; s2 = 'long';
>> strcmp(s1,s2)
ans =
     0
```

```
>> doc strcmp
>> doc strcmpi
```

case-insensitive

Useful String Functions

`sprintf`	-	write formatted string
`sscanf`	-	read formatted string
`deblank`	-	remove trailing blanks
`strcmp`	-	compare strings
`strcmpi`	-	compare strings
`strmatch`	-	find possible matches
`upper`	-	convert to upper case
`lower`	-	convert to lower case
`blanks`	-	string of blanks
`strjust`	-	left/right/center justify string
`strtrim`	-	remove leading/trailing spaces
`strrep`	-	replace strings
`findstr`	-	find one string within another

Structures

Structures have named 'fields' that can store all kinds of data: vectors, matrices, strings, cell arrays, other structures

`name.field`

```
student.name = 'Apple, A.';              % string
student.id = 12345;                      % number
student.exams = [85, 87, 90];            % vector
student.grades = {'B+', 'B+', 'A'};      % cell
```

```
>> student
student =
      name: 'Apple, A.'
        id: 12345
     exams: [85 87 90]
    grades: {'B+' 'B+' 'A'}
```

```
>> class(student)
ans =
struct
```

structures can also be created with **`struct()`**

Structure Arrays

add two more students with partially defined fields,
rest of fields are still empty

structure array index

```
student(2).name = 'Twitter, T.';
student(3).id = 345678;
```

```
>> student
student =
1x3 struct array with fields:
    name
    id
    exams
    grades
```

Structure Arrays

```
>> student(1)
ans =
        name: 'Apple, A.'
          id: 12345
       exams: [85 87 90]
      grades: {'B+' 'B+' 'A'}
```

```
>> student(2)
ans =
        name: 'Twitter, T.'
          id: []
       exams: []
      grades: []
```

```
>> student(3)
ans =
        name: []
          id: 345678
       exams: []
      grades: []
```

Structure Arrays

the missing field values can be defined later and don’t have to be of the same type or length as those of the other entries

```
>> student(3).exams = [70 80];

>> student(3)
ans =
    name: []
      id: 345678
   exams: [70 80]
  grades: []
```

Accessing Structure Elements

```
>> student(1)
ans =
      name: 'Apple, A.'
        id: 12345
     exams: [85 87 90]
    grades: {'B+' 'B+' 'A'}
```

```
>> student(1).name(5)     % ans =
                          % e
>> student(1).exams(2)    % ans =
                          %    87
>> student(1).grades(3)   % ans =
                          %   'A'
>> student(1).grades{3}   % ans =
                          % A
```

cell vs. content indexing

```
s.a = [1 2; 3 4];
s.b = {'a', 'bb'; 'ccc', 'dddd'};

>> s
s =
    a: [2x2 double]
    b: {2x2 cell}
```

```
>> s.a
ans =
     1     2
     3     4

>> s.a(2,2)
ans =
     4
```

```
>> s.b
ans =
    'a'      'bb'
    'ccc'    'dddd'

>> s.b(2,1), s.b{2,1}
ans =
     'ccc'
ans =
ccc
```

cell vs. content indexing

Nested Structures

```
student.name = 'Apple, A.';
student.id = 12345;
student.work.exams = [85, 87, 90];
student.work.grades = {'B+', 'B+', 'A'};

>> student
student =
    name: 'Apple, A.'
      id: 12345
    work: [1x1 struct]

>> student.work          % sub-structure
ans =
     exams: [85 87 90]
    grades: {'B+'  'B+'  'A'}
```

Stucture Functions

struct	-	create structure
fieldnames	-	get structure field names
isstruct	-	test if a structure
isfield	-	test if a field
rmfield	-	remove a structure field
struct2cell	-	convert structure to cell array

```
>> C = struct2cell(student)
C =
    'Apple, A.'
    [     12345]
    [1x1 struct]
>> C{3}
ans =
     exams: [85 87 90]
    grades: {'B+'  'B+'  'A'}
```

content indexing

Cell Arrays

Like structures, cell arrays are containers of all kinds of data: vectors, matrices, strings, structures, other cell arrays, functions.

A cell is created by putting different types of objects in curly brackets { }, e.g.,

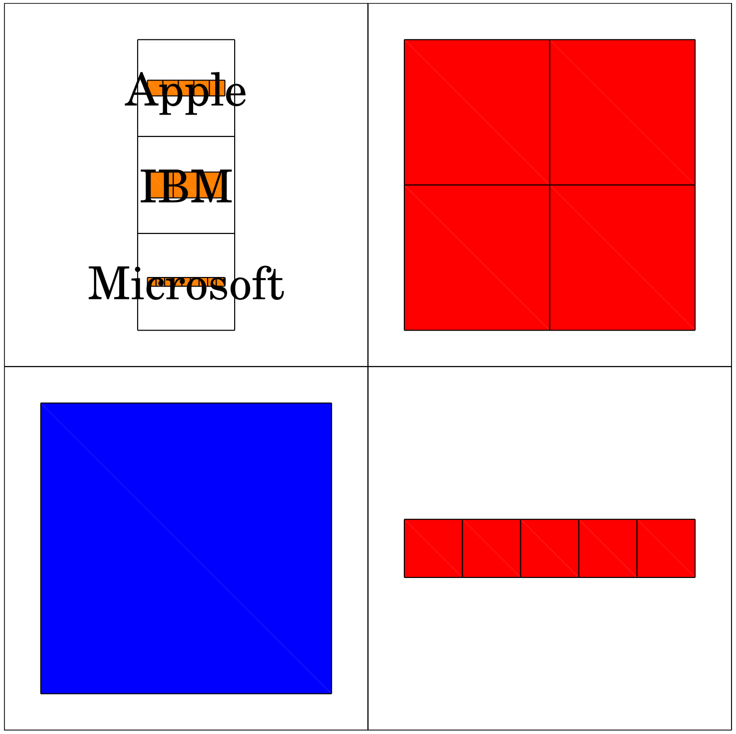
```
c = {A, B, C, D};      % 1x4 cell
c = {A; B; C; D};      % 4x1 cell
c = {A, B; C, D};      % 2x2 cell
```

where **A,B,C,D** are arbitrary objects

c{i,j} accesses the data in **i,j** cell
c(i,j) is the cell in the **i,j** position

cell vs. content indexing

```
A = {'Apple'; 'IBM'; 'Microsoft'};   % cells
B = [1 2; 3 4];                       % matrix
C = @(x) x.^2 + 1;                    % function
D = [10 20 30 40 50];                 % row

c = {A,B;C,D}   % define 2x2 cell array

c =
  {3x1 cell}  [2x2 double]
  @(x)x.^2+1  [1x5 double]

>> cellplot(c);
```



```
A = {'Apple'; 'IBM'; 'Microsoft'}

A =
    'Apple'
    'IBM'
    'Microsoft'
```

```
>> size(A), class(A)
ans =
     3     1
ans =
cell
```

comparing cell arrays of strings vs. strings

```
S = char('Apple', 'IBM', 'Microsoft')

S =
Apple
IBM
Microsoft
```

```
>> size(S), class(S)
ans =
     3     9
ans =
char
```

```
>> A(2), class(A(2))
ans =
    'IBM'
ans =
cell
```

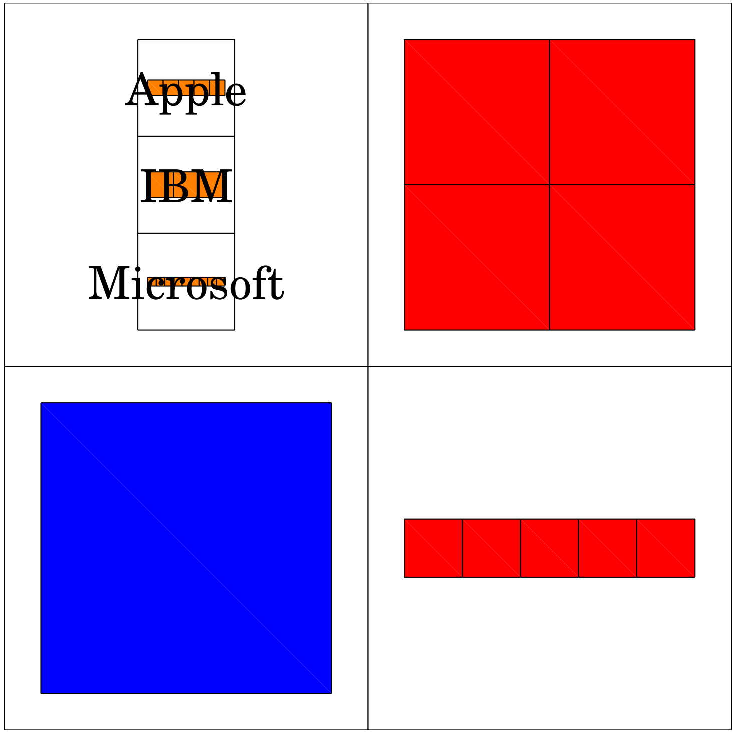
```
>> A{2}, class(A{2})
ans =
IBM
ans =
char
```

cell vs. content indexing

```
>> A'
ans =
   'Apple'   'IBM'   'Microsoft'
```

```
>> S

Apple
IBM
Microsoft

>> S'

AIM
pBi
pMc
l r
e o
  s
  o
  f
  t
```

```
A = {'Apple'; 'IBM'; 'Microsoft'};  % cells
B = [1 2; 3 4];                     % matrix
C = @(x) x.^2 + 1;                  % function
D = [10 20 30 40 50];               % row

c = {A,B;C,D}   % define 2x2 cell array

c =
  {3x1 cell}  [2x2 double]
  @(x)x.^2+1  [1x5 double]

>> cellplot(c);
```



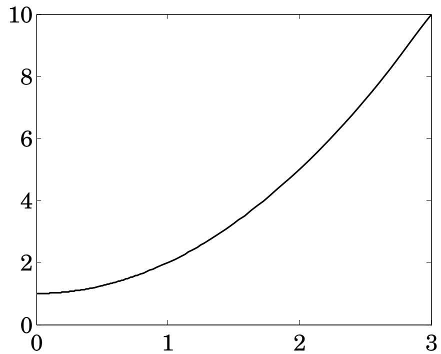
```
>> celldisp(c)

c{1,1}{1} =
Apple
c{1,1}{2} =
IBM
c{1,1}{3} =
Microsoft

c{2,1} =
    @(x)x.^2+1

c{1,2} =
     1     2
     3     4

c{2,2} =
    10    20    30    40    50
```

content indexing with { }

```
>> c{1,1}
ans =
    'Apple'
    'IBM'
    'Microsoft'

>> c{2,1}
ans =
    @(x)x.^2+1
```

content indexing with { }

```
>> c{1,1}{3}
ans =
Microsoft
>> c{1,1}{3}(6)
ans =
s

>> x = [1 2 3];
>> c{2,1}(x)
ans =
     2     5    10

>> fplot(c{2,1},[0,3]);
```




```
>> c{1,2}(2,:)
ans =
     3     4

>> c{1,2}(1,2)
ans =
     2

>> c{2,2}(3)
ans =
    30
```

cell indexing ( )
content indexing { }

```
>> c(2,2)
ans =
    [1x5 double]
```

cell

```
>> class(c(2,2))
ans =
cell
```

```
>> c{2,2}
ans =
    10      20     30      40      50
```

cell contents

```
>> class(c{2,2})
ans =
double
```

cell indexing ( )
content indexing { }

```
>> c{1,1}(2)
ans =
    'IBM'
```

cell

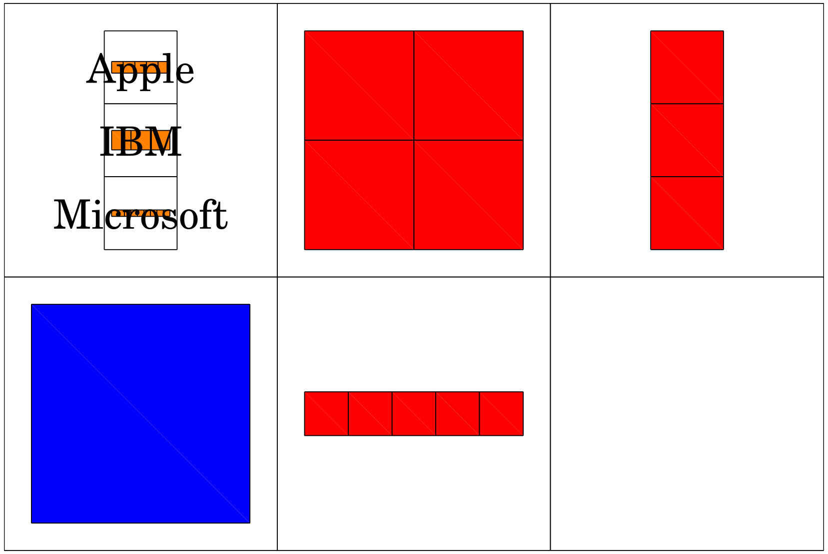
```
>> class(c{1,1}(2))
ans =
cell
```

```
>> c{1,1}{2}
ans =
IBM
```

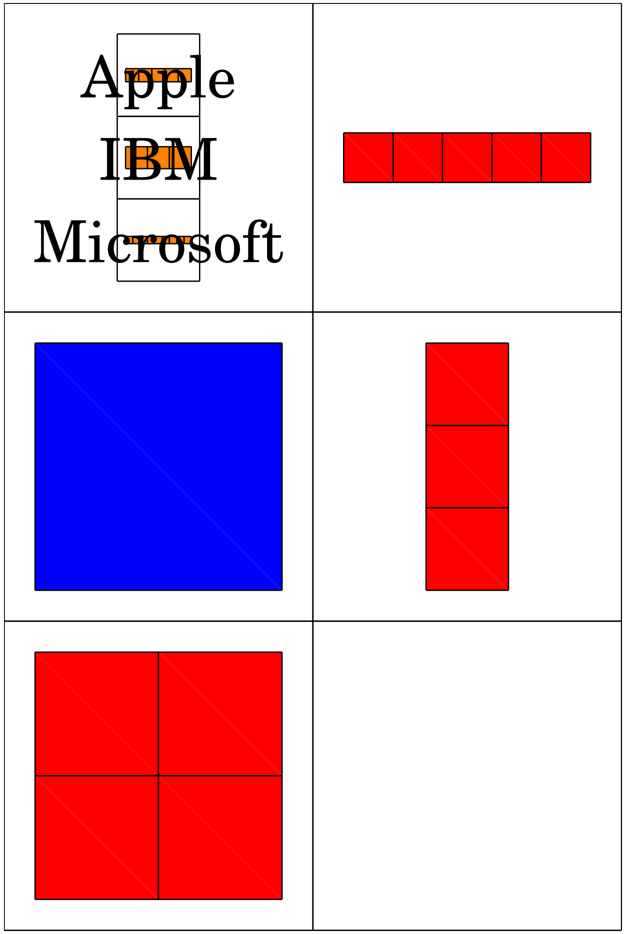
cell contents

```
>> class(c{1,1}{2})
ans =
char
```

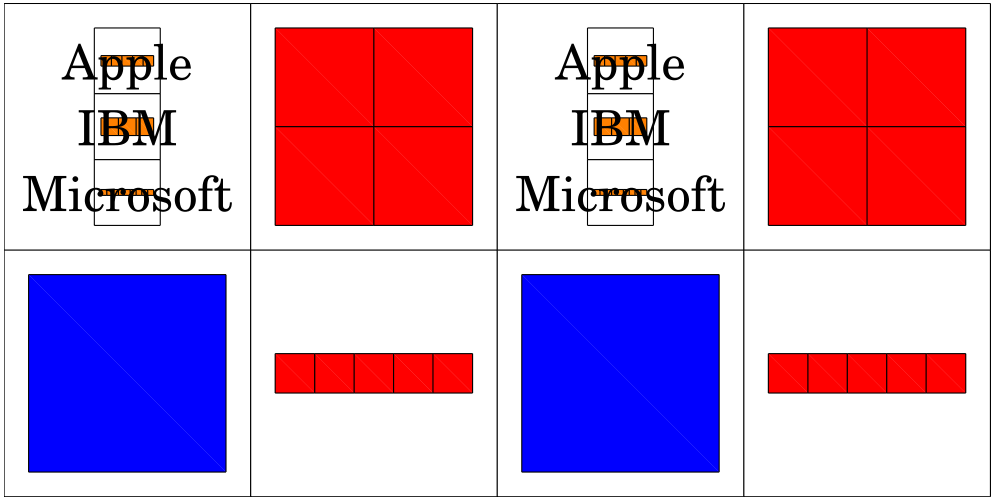
```
>> d = c;

>> % d(1,3) = {[4 5 6]'};    % define as cell
>> d{1,3} = [4,5,6]'         % define content

d =
  {3x1 cell}  [2x2 double]  [3x1 double]
  @(x)x.^2+1  [1x5 double]            []

>> d(2,3)
ans =
    {[]}

>> d{2,3}
ans =
     []

>> cellplot(d);
```



```
>> e = reshape(d,3,2)

>> cellplot(e)
```




```
>> f = repmat(c,1,2)

>> cellplot(f)
```



try also

```
>> f = d';
```

changing cell array contents

```
>> c{1,1}{2} = 'Google';
>> c{1,2} = 10*c{1,2};
>> c{2,2}(3) = 300;
>> celldisp(c)
c{1,1}{1} =
Apple
c{1,1}{2} =
Google
c{1,1}{3} =
Microsoft
c{2,1} =
    @(x)x.^2+1
c{1,2} =
    10    20
    30    40
c{2,2} =
    10    20    300    40    50
```

could have used:

```
c{1,1}(2) = {'Google'};
```

why not ?

```
c(1,2) = 10*c(1,2);
```

why not ?

```
c{2,2}{3} = 300;
```

varargin
varargout

varargin,varargout are cell arrays that allow the passing a variable number of function inputs & outputs

```
% [x,y,vx,vy] = trajectory(t,v0,th0,h0,g)

function [varargout] = trajectory(t,v0,varargin)

Nin = nargin-2;   % number of varargin inputs

if Nin==0, th0=90; h0=0; g=9.81; end
if Nin==1, th0=varargin{1}; h0=0; g=9.81; end
if Nin==2, th0=varargin{1}; ...
           h0=varargin{2}; g=9.81; end
if Nin==3, th0=varargin{1}; ...
           h0=varargin{2}; g=varargin{3}; end
```

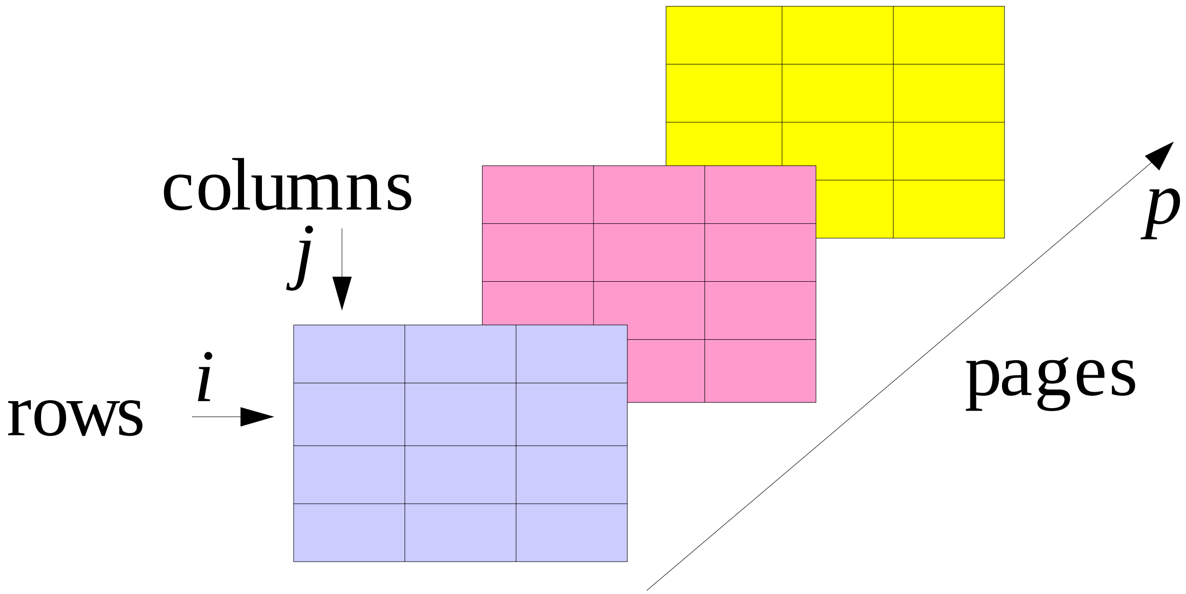
continues

```
th0 = th0 * pi/180;    % convert to radians

x = v0 * cos(th0) * t;
y = h0 + v0 * sin(th0) * t - 1/2 * g * t.^2;
vx = v0 * cos(th0);
vy = v0 * sin(th0) - g * t;

if nargout==1; varargout{1} = x; end
if nargout==2; varargout{1} = x; ...
               varargout{2} = y; end
if nargout==3; varargout{1} = x;
               varargout{2} = y; ...
               varargout{3} = vx; end
if nargout==4; varargout{1} = x; ...
               varargout{2} = y; ...
               varargout{3} = vx; ...
               varargout{4} = vy; end
```

Multidimensional Arrays

A three-dimensional array is a collection of two-dimensional matrices of the same size, and are characterized by triple indexing, e.g., **`A(i,j,p)`** is the (i,j) matrix element of the p-th matrix.

Higher-dimensional arrays can also be defined, e.g., a 4D array is a collection of 3D arrays of the same size.



applications in video processing

```
>> a = [1 2; 3 4];
>> A(:,:,1) = a;
>> A(:,:,2) = 10*a;
>> A(:,:,3) = 100*a;

>> A
A(:,:,1) =
     1     2
     3     4

A(:,:,2) =
    10    20
    30    40

A(:,:,3) =
   100   200
   300   400
```

pages

`sum, min, max`
can operate along the i,j,p dimensions

```
A(:,:,1) =
     1     2
     3     4

A(:,:,2) =
    10    20
    30    40

A(:,:,3) =
   100   200
   300   400
```

```
>> sum(A,3)
ans =
   111   222
   333   444
```

```
>> sum(A,1)
ans(:,:,1) =
     4     6
ans(:,:,2) =
    40    60
ans(:,:,3) =
   400   600
```

```
>> sum(A,2)
ans(:,:,1) =
     3
     7
ans(:,:,2) =
    30
    70
ans(:,:,3) =
   300
   700
```

```
A(:,:,1) =
     1     2
     3     4


A(:,:,2) =
    10    20
    30    40


A(:,:,3) =
   100   200
   300   400
```

```
>> min(A,[],3)
ans =
     1     2
     3     4
```

```
>> min(A,[],1)
ans(:,:,1) =
     1     2
ans(:,:,2) =
    10    20
ans(:,:,3) =
   100   200
```

```
>> min(A,[],2)
ans(:,:,1) =
     1
     3
ans(:,:,2) =
    10
    30
ans(:,:,3) =
   100
   300
```

```
A(:,:,1) =
     1     2
     3     4

A(:,:,2) =
    10    20
    30    40

A(:,:,3) =
   100   200
   300   400
```

```
>> A>20 & A<300
ans(:,:,1) =
     0     0
     0     0
ans(:,:,2) =
     0     0
     1     1
ans(:,:,3) =
     1     1
     0     0

>> k = find(A>20 & A<300)
k =
     6
     8
     9
    11
```

column-order across pages